Chapter 11

Radial Basis Functions

11.1 Polynomial Regression

In the interpolation problem of Chapter 1, we saw that a polynomial of degree $n-1$ will connect n data points. The regression problem is different in that we are given n data points, and we are asked to fit a polynomial of degree k to it. There now need be no connection between the degree and the number of data points, and typically, we have many more points than degrees.

1. **Remark:** We begin by assuming that the function

$$f : X \subset \mathbb{R} \to Y \subset \mathbb{R}$$

is of degree k, and let the number of data points be p. Form the partial Vandermonde matrix (size will be $p \times (k+1)$), so that we are solving the matrix equation:

$$VC = Y$$

where the i^{th} row of V is:

$$(x_i^k \ x_i^{k-1} \ \ldots x_1 \ 1) \tag{11.1}$$

2. **Exercise:** Find the least squares solution to the 1-d polynomial regression problem by using the SVD and the generalized inverse.

3. **Exercise:** Rewrite Matlab's `vander` command so that `Y=vander1(X,k)` where X is $m \times 1$ (m is the number of points), $k-1$ is the degree of the polynomial and Y is the $m \times k$ matrix whose rows look like those in Equation 11.1.

4. **Remark:** Polynomials seldom do a good job at regression, and the following exercises show this.

5. **Exercise:** Load the data in the file `poly_regres.mat`. Here you will find data sets P and T, where P are points uniformly distributed between -1 and 1, and T are the desired targets (thus, the desired graph are points (p_i, t_i)). Note that P and T are given as rows, but will need to be columns in our suite of commands. Look at the following sequence of commands, and reproduce them. Be sure you understand what each one does.

   ```
   Y=-1:0.01:1;
   TT=T+0.1*randn(size(T));
   A=vander1(P',18);
   G=A\TT';
   Z=polyval(G,Y');
   plot(P,T,P,TT,'*',Y,Z);
   ```

   In this example, we used a polynomial of degree 17. Now try the same thing with:

   (a) Use the original targets, with degrees 10, 50, 100.
   (b) Use the noisy targets with degrees 10, 50, 100.

   Which polynomials work "best"?

6. **Remark:** A second order polynomial of 1 variable has the following basis terms: $\{1, x, x^2\}$. A second order polynomial of 2 variables has: $\{1, x, y, xy, x^2, y^2\}$. Going to dimension 3: $\{1, x, y, z, xy, xz, yz, x^2, y^2, z^2\}$. Recall that for each basis term, there is an associated unknown constant. This unknown is called a parameter, and the number of parameters will be the number of columns in the regression matrix V.

7. **Exercise:** Show that for a second order polynomial in dimension n, there are:
   $$\frac{n^2 + 3n + 2}{2} = \frac{(n+2)!}{2!n!}$$
   unknown constants, or parameters.

8. **Remark:** In general, if we use a polynomial of degree d in dimension n, there are
   $$\frac{(n+d)!}{d!\, n!} \approx n^d$$
   parameters. How many parameters do we have if we wish to use a polynomial of degree 10 in dimension 5?

9. **Exercise:** Write a Matlab subroutine for which we input data in $\mathbb{R}^4$ and outputs the regression matrix for a second order approximation. Solve the regression problem for the data in the file "`polydat.mat`".

10. **Remark:** This explosion in the number of parameters needed for regression is known as **the curse of dimensionality**.

11. **Exercise:** Suppose that we have a unit box in n dimensions (corners lie at the points $(\pm 1, \pm 1, \ldots, \pm 1)$), and we wish to subdivide each edge by cutting it into k equal pieces. Verify that there are k^n sub-boxes. Consider what this number will be if k is any positive integer, and n is conservatively large, say $n = 1000$! This rules out any simple, general, "look-up table" type of solution to the regression problem.

12. **Definition:** Let $\{f_i\}_{i \in \mathcal{I}}$ be a set of real-valued functions whose domain is $[0, 1]^n$. This set forms a *basis* for the set of continuous functions on $[0, 1]^n$ if the functions are linearly independent and the set is *dense*. If that is the case, we say that it is possible to formulate any continuous function g in terms of the basis:
$$g(x) = \sum_{i \in \mathcal{I}} \alpha_i f_i(x)$$

13. **Example:** The Stone-Weierstrasse Theorem says that the set $\{x^n\}_{i=1}^{\infty}$ is dense in $C[0, 1]$. This is the polynomial approximation that we've been doing. Notice that the set of basis functions did not depend on the data (although, of course, the coefficients did).

14. **Example:** The Fourier Basis is dense for 2π periodic functions, or on $C[0, 2\pi]$. The Fourier Basis is given by: $\{\sin(nx), cos(mx)\}_{m,n=1}^{\infty}$

15. **Definition:** A set of basis functions for $C[a, b]^n$ is said to be *fixed* if the basis does not depend on the data.

16. **Remark:** Recent results [2, 3] have shown that *any* fixed basis will suffer from the curse of dimensionality. This leads us to consider basis functions that are *semi-analytic* in that the basis functions we choose will be data-dependent. Ideally, the number of parameters needed for regression should depend not on the dimension of the data, but on the complexity of the function.

11.2 Distance Matrices

We will begin considering a type of regression model known as "Radial Basis Functions" shortly. Before we do that, let us consider a special type of matrix that we will be using extensively.

1. **Definition: The Euclidean Distance Matrix (EDM)** Let $\{\boldsymbol{x}^{(i)}\}_{i=1}^{P}$ be a set of points in $\mathbb{R}^n$. Let the matrix $\boldsymbol{A}$ be the P by P matrix so that the $(i, j)^{\text{th}}$ entry of $\boldsymbol{A}$ is given by:
$$A_{ij} = \|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\|_2$$
Then $\boldsymbol{A}$ is the Euclidean Distance Matrix (EDM) for the data set $\{\boldsymbol{x}^{(i)}\}_{i=1}^{P}$.

2. **Exercise:** By hand, construct the EDM for the data (given as $n \times p$, where n is the dimension, and p is the number of points):

(a) $X = (1, 2, 3, 4)$

(b) $X = \begin{pmatrix} -1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

(c) $X = \begin{pmatrix} -1 & 1 & 1 & -1 \\ -1 & -1 & 1 & 1 \end{pmatrix}$

3. **Exercise:** Print and analyze the Matlab function EDM. Be sure you understand what it (and the functions it calls) are doing.

4. **Remark:** The EDM is a symmetric matrix with 0's along its diagonal.

5. **Exercise:** What can we say about the eigenvalues and eigenvectors of the EDM?

6. **Matlab Comment:** One can find the minimum element in a vector: `min(x)` or in a matrix: `min(min(X))`

7. **Theorem (Schoenberg, 1937)**. Let $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_p\}$ be p distinct points in $\mathbb{R}^n$. Then the EDM is invertible.

8. **Remark:** This theorem implies that, if $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_p\}$ are p distinct points in $\mathbb{R}^n$, and $\{y_1, \ldots, y_p\}$ are points in $\mathbb{R}$, then there exists a function $f : \mathbb{R}^n \to \mathbb{R}$ such that:

$$f(\boldsymbol{x}) = \alpha_1 \|\boldsymbol{x} - \boldsymbol{x}_1\| + \ldots + \alpha_p \|\boldsymbol{x} - \boldsymbol{x}_p\|$$

and $f(\boldsymbol{x}_i) = y_i$. Therefore, this function f solves the interpolation problem.

9. **Exercise:** Verify that the α_i defined above are solved:

$$A\boldsymbol{\alpha} = Y$$

where $A_{ij} = \|\boldsymbol{x}_i - \boldsymbol{x}_j\|$ and Y is a column vector.

10. **Exercise:** This approximation is a generalization of a piecewise linear interpolation. To see this, try plotting

$$f(x) = -2|x + 1| + |x - 2| + \frac{1}{2}|x - 3|$$

(Hint: In Matlab, $|x| =$`abs(x)`).

11. **Remark:** As we saw in Chapter 2, the invertibility of a matrix depends on its smallest eigenvalue. A recent theorem states "how invertible" the EDM is:

12. **Theorem [?]:** Let $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_p\}$ be p distinct points in $\mathbb{R}^n$. Let ϵ be the smallest element in the EDM, so that

$$\|\boldsymbol{x}_i - \boldsymbol{x}_j\| \geq \epsilon,\ i \neq j$$

Then we have that all eigenvalues λ_i are all bounded away from the origin- there is a constant c so that:

$$\lambda_i \geq \frac{c\epsilon}{\sqrt{n}} \tag{11.2}$$

13. **Exercise:** Let's examine what this is saying by looking at random data in $\mathbb{R}^4$. Write the following script and plot the resulting x and y. For comparisons, plot x versus $x/2$. This will take a few minutes, so we print the "i" to see where the loop is. Your output should be similar to Figure 11.1.

```
for i=1:100
i
X=rand(50,4); %Randomly choose 50 points
A=edm(X);
V=eig(A);
x(i)=min(min(A+eye(50)));  %This is epsilon.
y(i)=min(abs(V));          %Smallest eigenvalue.
end
plot(x,y);
```

11.3 Radial Basis Functions

1. **Definition: The Transfer Function and Matrix** Let $\phi : \mathbb{R}^+ \to \mathbb{R}$ be chosen from the list below:

$$\begin{array}{rcll}
\phi(r,\sigma) & = & \exp\left(\frac{-r^2}{\sigma^2}\right) & \text{Gaussian} \\
\phi(r) & = & r^3 & \text{Cubic} \\
\phi(r) & = & r^2\log(r) & \text{Thin Plate Spline} \\
\phi(r) & = & \frac{1}{r+1} & \text{Cauchy} \\
\phi(r,\beta) & = & \sqrt{r^2+\beta} & \text{Multiquadric} \\
\phi(r,\beta) & = & \frac{1}{\sqrt{r^2+\beta}} & \text{Inverse Multiquadric}
\end{array}$$

Then ϕ is a *transfer function.* If we apply ϕ to a matrix elementwise so that:

$$\boldsymbol{\Phi} = (\phi(\boldsymbol{A}))_{ij} = (\phi(A_{ij}))$$

Then $\boldsymbol{\Phi}$ is the transfer matrix. Note that when A is a matrix, Matlab will automatically perform some functions elementwise- for example, `sin(A)`.

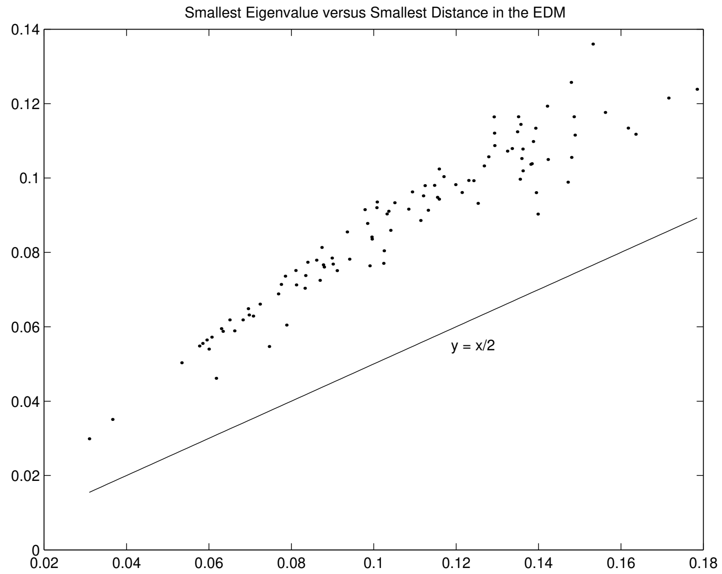

Figure 11.1: This plot illustrates the statement in Equation (11.2), which tells us how invertible the Euclidean Distance Matrix is.

2. **Example:** If $\phi(r) = \sqrt{r^2 + 1}$, then ϕ applied to the matrix A will be done via:

```
(A.^2+1).^0.5
```

3. **Remark:** There are other transfer functions one can choose. For a broader definition of transfer functions, see Micchelli [26]. We will examine the effects of the transfer function on the radial approximation shortly.

4. **Remark:** Matlab only uses a Gaussian transfer function. The next few exercises show some of the implications.

5. **Exercise:** Run Matlab's demos: `demorb1` and `demorb3` to see how σ effects the functional approximation. (You can just type "demo", then go to the Neural Network Toolbox demos, then choose one of the RBF demos- "underlapping" neurons, and "overlapping" neurons).

6. **Exercise:** We need to consider how different values of σ effects the width of the Gaussian. Use Maple to animate the Gaussian function:

```
with(plots):
animate(exp(-x^2/t^2),x=-5..5,t=0.5..2);
```

7. **Definition:** We'll make a working definition of the *width* of the Gaussian: It is the value a so that k percentage of the area is between $-a$ and a (so k is between 0 and 1). The actual value of k will be problem-dependent.

8. **Exercise:** We will approximate:

$$\left(\int_{-b}^{b} \mathrm{e}^{-x^2}\, dx\right)^2 = \int_{-b}^{b}\int_{-b}^{b} \mathrm{e}^{-(x^2+y^2)}\, dx\, dy \approx \int\int_{B} \mathrm{e}^{-(x^2+y^2)}\, dB$$

where B is the disk of radius b. Show that this last integral is:

$$\pi\left(1-\mathrm{e}^{-b^2}\right)$$

9. **Exercise:** Using the previous exercise, conclude also that:

$$\int_{-\infty}^{\infty} \mathrm{e}^{\frac{-x^2}{\sigma^2}}\, dx = \sigma\sqrt{\pi}$$

10. **Exercise:** Use the previous two exercises to show that our working definition of the "width" a, means that, given a we would like to find σ so that:

$$\int_{-a}^{a} \mathrm{e}^{\frac{-x^2}{\sigma^2}}\, dx \approx k\int_{-\infty}^{\infty} \mathrm{e}^{\frac{-x^2}{\sigma^2}}\, dx$$

11. **Exercise:** Show that the last exercise implies that, if we are given k and a, then we should take σ to be:

$$\sigma = \frac{a}{\sqrt{-\ln(1-k^2)}} \tag{11.3}$$

12. **Remark:** Matlab uses the following approximation for σ (See, for example, `designrb`, which is in the file `newrb.m`):

$$\sigma = \frac{a}{\sqrt{-\ln(0.5)}}$$

11.4 Interpolation via Radial Basis Functions

1. **Notation:** Let $\{\boldsymbol{x}^{(i)}\}_{i=1}^{P} \subset \mathbb{R}^N$. Suppose we also have a data set $\{\boldsymbol{y}^{(i)}\}_{i=1}^{P} \subset \mathbb{R}^M$. Let $\boldsymbol{Y}$ be the P by M matrix constructed so that its i^{th} row is $(\boldsymbol{y}^{(i)})^{\mathrm{T}}$.

2. **Definition:** The interpolation problem is to determine a function $\boldsymbol{F}$ such that

$$\boldsymbol{y}^i = \boldsymbol{F}(\boldsymbol{x}^{(i)}),\ i = 1 \dots P \tag{11.4}$$

3. **Remark:** So far, $\boldsymbol{F}$ is not well defined. Think about how many functions (continuous, not continuous, differentiable, not differentiable) will go through those points.

4. **Remark:** We will assume a certain form for $\boldsymbol{F}$ so that Equation (11.4) has the least squares solution.

5. **Remark:** Assume $\boldsymbol{F}$ is defined so that the j^{th} element of $\boldsymbol{F}(\boldsymbol{x}^{(i)})$ is given by:

$$\boldsymbol{F}_j(\boldsymbol{x}^{(i)}) = \sum_{k=1}^{P} \omega_{kj}\ \phi\left(\|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(k)}\|\right) + b_j$$

where the ω_{ij} are parameters to be determined, where i runs from 1 to P, and j runs from 1 to m. Then the interpolation problem translates to the following matrix equation, for which we need to solve for the constants $\omega_{ij} \in \Omega$, which is a P by M matrix.

$$\boldsymbol{\Phi}\Omega + \boldsymbol{b} = \boldsymbol{Y} \tag{11.5}$$

By the properties listed above, if $\boldsymbol{b} = \boldsymbol{0}$, then this equation has a unique solution

$$\Omega = \boldsymbol{\Phi}^{-1}\boldsymbol{Y}$$

since $\boldsymbol{\Phi}$ is nonsingular, and ϕ has been chosen from the classes allowed.

6. **Exercise:** Show that Equation (11.5) can be written as a single matrix equation:

$$\hat{\boldsymbol{\Phi}}\hat{\Omega} = \boldsymbol{Y}$$

so that we can also solve for the vector $\boldsymbol{b}$. Hint: Think about the sizes of the matrices.

7. **Definition:** A *training set* is a set of data that is used for computing the weights and biases of a model. The *validation set* is a different set of data that is used to test the performance of the model found using the training set. We'll see why these are important in the next section.

11.5 Generalization

We saw earlier with polynomials that many times we do not desire an interpolative solution. We will be willing to trade some accuracy for a smoother, less complex function approximation.

1. **Ill-Defined Problems:** Not all problems have solutions. We may know this ahead of time, but wish to find a solution that is "close". For example, we know that there is a fundamental difference between $\mathbb{R}^n$ and $\mathbb{R}^m$, and so there may be problems in finding functions that map one to another.

2. **Example:** Consider the following example. The object in Figure 11.2 is not one dimensional, and so there does not exist a continuous function from $\mathbb{R}^1$ to the object presented. However, as soon as we discretize the object to data points, then we can interpolate the data. One solution is

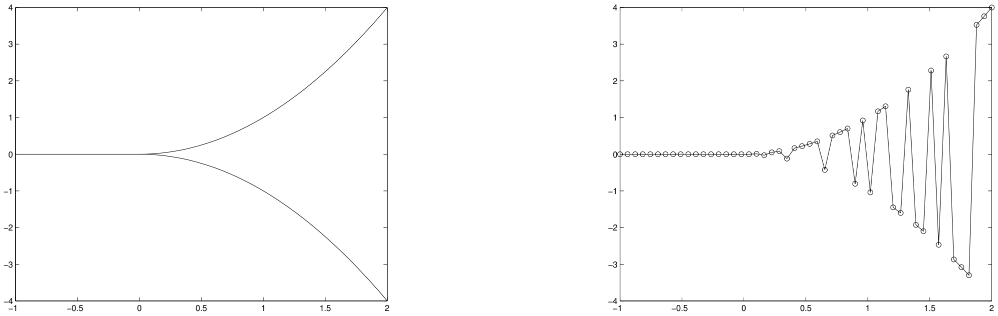

Figure 11.2: The image to the left is the original range set. The image to the right depicts the RBF solution to the discretized range. This shows that, while the RBF can obtain an exact solution to the interpolation problem, that solution is often not the optimal solution to generalize the range.

presented in Figure 11.2. Note that the best (in terms of least squares error) one dimensional representation of this object should be the zero function.

3. **Remark:** From [4], we obtain an extension of the RBF as stated earlier. Rather than computing the full EDM, we replace the model of the function $\boldsymbol{F}$:

$$\boldsymbol{F}_j(\boldsymbol{x}^{(i)}) = \sum_{k=1}^{K} \omega_{kj}\ \phi\left(\|\boldsymbol{x}^{(i)} - \boldsymbol{c}_k\|\right) + b_j$$

where we will refer to the set $\{\boldsymbol{c}_k\}_{k=1}^{K}$ as the **set of centers**.

4. **Remark:** Given a set of $K << P$ centers, we redefine the transfer matrix so that

$$(\boldsymbol{\Phi}_c)_{ij} = \phi\left(\|\boldsymbol{x}^{(i)} - \boldsymbol{c}_j\|\right)$$

and $\boldsymbol{\Phi}_c$ is now a P by K matrix.

5. **Remark:** Once a set of centers has been chosen, then we have changed the interpolation problem to a problem of finding the least squares solution to:

$$\hat{\boldsymbol{\Phi}}_c \hat{\Omega} = \boldsymbol{Y}$$

whose solution is:

$$\hat{\Omega} = \hat{\boldsymbol{\Phi}}_c^{+} \boldsymbol{Y} \tag{11.6}$$

where $\hat{\boldsymbol{\Phi}}_c^{+}$ is the Moore-Penrose generalized inverse of $\hat{\boldsymbol{\Phi}}_c$.

6. **Remark:** We will be using only this new form of the transfer matrix, and so will suppress the "c" subscripts for the remainder of this chapter.

7. **Summary thus far:** We have introduced the RBF equations, and the role of the EDM. We have now reduced the size of the EDM via the use of centers. Once the set of centers has been chosen, we can solve the RBF equation using the pseudo-inverse. That is, solve Equation (11.6). In Matlab, we can use either the SVD, the backslash command, or the "pinv" command.

   Therefore, in using an RBF we must answer 2 questions:

   (a) What should the transfer function be? If it has an extra parameter (like the Gaussian), what should it be? A rule of thumb for the Gaussian: The width should be wider than the distance between data points, but smaller than the diameter of the set.

   (b) How many centers should I use, and where should I put them?

   Once these questions are answered, we can solve the problem

8. **Remark:** For the first question, there is no general answer, and we've examined some issues related to which transfer function to use. In theory, it doesn't matter, as long as the data is scaled appropriately. In practice, the choice of transfer function may force you to rescale your data, and may impact on the number of centers needed.

9. **Remark:** On the number of centers, we need to remember the tradeoff between generalization and memorization. In general, we want the fewest number of centers necessary to attain our error goal on the *Training Set* while still getting a good error on the *Validation Set*. In most problems, these goals will be contradictory.

10. **Remark:** If we use only 1 center, we'll get some really bad errors. If we use the entire data set for the centers, the error on the training set will be zero. Somewhere in between we must be getting a good tradeoff, as Figure 11.3 depicts.

11. **Remark:** On the placement of centers, we have again some options. They are listed below, then we will consider each seperately:

    (a) Choose the centers at random.

    (b) Choose the centers using Linear Algebra (Orthogonal Least Squares).

    (c) Choose the centers using Nonlinear Optimization.

12. **Exercise:** We examine the question: If we fix the number of centers to be k, will one random choice of centers be as good as any other?

    Construct a training set of 100 points using $y = \sin(2\pi x) + \epsilon$, where ϵ is normally distributed noise with std 0.1. Fix the number of centers at 30.

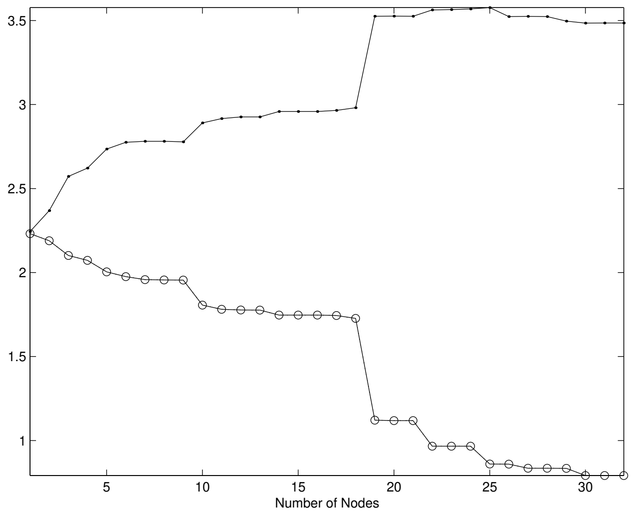

Figure 11.3: This is a plot of the error over the training set (plotted using circles) and the error over the validation set (plotted using dots). Using this picture, the best error possible is the point at which these curves intersect, which is using 1 center.

We're going to want to choose centers at random from the domain set. How should we code it? Remember- we do not want to choose 2 indices to be the same! See the sample code `randr.m`

Now, construct the RBF approximations using the cubic transfer function. We want to consider the error of the approximation. Use the function `rbfDesig.m`.

Construct a new data set using 1000 points. Compare the "true output" versus the RBF output on the data. Compute the mean squared error.

Repeat 25 times for different choices of the 30 centers and plot the 25 errors. What is your conclusion?

13. **Remark:** In using nonlinear optimization, the weights ω_{ij} and biases b_j are updated slowly so that the overall mean squared error gets smaller and smaller. If E is the mean squared error, then the update rule is:

$$\omega_{ij} = \omega_{ij} + h\frac{\partial E}{\partial \omega_{ij}}$$

and

$$b_j = b_j + h\frac{\partial E}{\partial b_j}$$

and we can even update the center locations:

$$\boldsymbol{c}_k = \boldsymbol{c}_k + h\frac{\partial E}{\partial \boldsymbol{c}_k}$$

14. **Remark:** Although these methods are much more complex in implementation than OLS (below), they are used extensively when the training data are not known all at once.

11.6 Orthogonal Least Squares

The following is a summary of the work in the reference [5]. We present the multidimensional extension that is the basis of the method used in Matlab's subfunction: `designrb`, which can be found in the file `newrb`. (Hint: To find the file, in Matlab type `which newrb`). We go through this algorithm in detail so that we can modify it to suit our needs. Unfortunately, Matlab's equations are transposed from our setup, so we will include the Matlab versions in regular type.

1. **Basic Idea:**

   (a) Notation Setup: Let $\boldsymbol{\Phi}$ be the full interpolation matrix, where we index the p columns:
   $$\boldsymbol{\Phi} = [\Phi_1, \ldots, \Phi_p]$$
   Let the domain be in $\mathbb{R}^n$ and the range be data in $\mathbb{R}^m$. Form the modified matrix:
   $$\hat{\boldsymbol{\Phi}} = [\Phi_1, \ldots, \Phi_p, \texttt{ones(p,1)}]$$
   so that the corresponding coefficient matrix Ω will now contain the *weights* w_{ij} (a $p \times m$ matrix) and *biases* $\boldsymbol{b}$ (an m-vector):
   $$\hat{\Omega} = \begin{bmatrix} \Omega \\ \boldsymbol{b}^T \end{bmatrix}$$

   (b) Suppose that we begin by assuming that all data points are candidate centers. We look at the RBF equation:
   $$\hat{\boldsymbol{\Phi}}\hat{\Omega} = \boldsymbol{Y}$$
   where $\boldsymbol{Y}$ is a $p \times m$ matrix.

   (c) Think of the columns of $\boldsymbol{\Phi}$ as spanning one dimensional subspaces of $\mathbb{R}^p$. Then the columns of $\boldsymbol{Y}$ also form subspaces of $\mathbb{R}^p$.

   (d) We will choose that subspace of $\boldsymbol{\Phi}$ that most closely spans the column space of $\boldsymbol{Y}$ (as vectors in $\mathbb{R}^p$).

2. Algorithm: The basic algorithm proceeds as follows:
   (a) Look for the column of $\mathbf{\Phi}$ that most closely points in the same direction as a column of $\boldsymbol{Y}$.
   (b) Take that column out of $\mathbf{\Phi}$ (which makes $\mathbf{\Phi}$ smaller), then deflate the remaining columns. Another way to say this is that we remove the component of the columns of $\mathbf{\Phi}$ that point in the direction of our "winner".
   (c) Repeat.
3. **Remark:** Before continuing, let's review the linear algebra that we'll need to perform the operations listed above.
4. **Exercise:** Show that, if we have a set of vectors $X = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k]$ and a vector $\boldsymbol{y}$, then the vector in X that most closely points in the direction of $\boldsymbol{y}$ (or $-\boldsymbol{y}$) is found by computing the maximum of:
$$\left(\frac{\boldsymbol{x}_i^T \boldsymbol{y}}{\|\boldsymbol{x}_i\|\|\boldsymbol{y}\|}\right)^2 \tag{11.7}$$
5. **Exercise:** First, recall that:
$$\frac{\boldsymbol{v}\boldsymbol{v}^T}{\|\boldsymbol{v}\|^2} \tag{11.8}$$
is an orthogonal projector to the one dimensional subspace spanned by $\boldsymbol{v}$. Using this, show that the following set of Matlab commands will return a matrix whose columns are in the orthogonal complement of the subspace spanned by $\boldsymbol{v}$. That is, $\boldsymbol{v}$ is orthogonal to the columns of the final matrix in the computation:
```
a=v'*P/(v'*v);
P=P-v*a;
```
6. **Remark:** There are some other interesting programming elements in the Matlab suite of radial basis functions. Below, we examine some Matlab "tricks" that will come in handy.
7. **Exercise:** Let X and Y be matrices of the same size, and x and y be vectors. Then explain what each line of commands will do:
   (a) `X(:,3)=[];`
   (b) `x(2)=[];`
   (c) `X=[X Y(:,3)];`
   (d) `x=[x y(2)];`
   (e) `I=1:2:5;  X=Y(:,I);`
   (f) `[m,n]=size(X); X=[X;ones(1,N)];`

(g) `A=X./Y;`

8. **Exercise:** Let p be a $n \times k$ matrix whos *columns* are data points $\boldsymbol{x}^{(i)} \in \mathbb{R}^n$. Let a be the desired spread of the Gaussian (See Equation 11.3). Verify that the (i,j) component of the following Matlab commands is

$$e^{\frac{-\|\boldsymbol{x}^{(i)}-\boldsymbol{x}^{(j)}\|^2}{\sigma^2}}$$

```
b=sqrt(-log(0.5))/a;
P=radbas(dist(p',p)*b);
```

9. **Exercise:** Show that, if $X = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k]$ are column vectors, then

$$\texttt{sum(X.*X)=}\left(\|\boldsymbol{x}_1\|^2, \ldots, \|\boldsymbol{x}_k\|^2\right)$$

10. **Exercise:** Write a Matlab function, `mnormal` that will input a matrix, and output the matrix with normalized columns.

11. **Exercise:** Let $X = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m]$ and $Y = [\boldsymbol{y}_1, \ldots, \boldsymbol{y}_k]$. Show that the (i,j) component of `X'*Y` is given by:

$$\boldsymbol{x}_i^T \boldsymbol{y}_j$$

12. **Exercise:** Verify that the following Matlab commands solves the (least squares) equation: $wp + b = t$ for w and b:

```
[pr,pc]=size(p);
x=t/[p;ones(1,pc)];
w=x(:,1:pr);
b=x(:,pr+1);
```

13. **Exercise:** What does the following set of Matlab commands do? (Assume M is a matrix)

```
replace=find(isnan(M));
M(replace)=zeros(size(replace));
```

14. **Remark:** We are now ready to examine the Matlab function `newrb` and its main subfunction, `designrb`. These programs use the OLS algorithm [5] described previously. This algorithm will tell us the columns of $\boldsymbol{\Phi}$ that are most critical to the approximation, and therefore will tell us which data points to use as centers.

15. **Matlab Comment:** The function `newrb` is basically a wrapper for the function (included in newrb) `designrb`. The introductory programming lines in `newrb` sets up the RBF neural network, then calls `designrb` to train it.

16. **Remark:** As a programming note, we will be selecting columns from $\mathbf{\Phi}$ and holding them in a matrix W. At this point, you may want to print the function `newrb.m` for reference. The algorithm below is an explanation of it.

Algorithm 11.6.1 *Orthogonal Least Squares for RBFs*

The function design is to input the domain and range (p, t)[1], the error goal and gaussian spread (`eg`, `sp`), and output the centers `w1`, σ's (actually, $\frac{1}{\sigma}$) are stored in `b1`, the coefficients ω_{ij} $\in$`w2` and the biases b_j $\in$`b2`. The other output arguments are optional.

```
function [w1,b1,w2,b2,k,tr] = designrb(p,t,eg,sp)

[r,q] = size(p);
[s2,q] = size(t);
b = sqrt(-log(.5))/sp;

% RADIAL BASIS LAYER OUTPUTS
P = radbas(dist(p',p)*b);
PP = sum(P.*P)';
d = t';
dd = sum(d.*d)';
```

17. **Exercise:** For each matrix above, write down the size (in terms of "number of points" and "dimension").

So far, we've constructed the interpolation matrix P=$\mathbf{\Phi}^T$, and initialized some of our variables. We're now ready to initialize the algorithm:

```
% CALCULATE "ERRORS" ASSOCIATED WITH VECTORS
e = ((P' * d)' .^ 2) ./ (dd * PP');
```

18. **Exercise:** What is the size of `e`? Show that the (j, k) component of `e` is given by (in terms of our previous notation):

$$\frac{\boldsymbol{\phi}_k^T \boldsymbol{y}_j}{\boldsymbol{y}_j^T \boldsymbol{y}_j \boldsymbol{\phi}_k^T \boldsymbol{\phi}_k}$$

In view of Equation 11.7, what does `e` represent?

In the following lines, there is some intriguing code. The integer `pick` is used to store our column choice. The vector `used` will be used to store the columns of $\mathbf{\Phi}$ that have been chosen, and the vector `left` are the columns remaining.

[1] Data is entered as "dimension" by "number of points" in Matlab, which is the transpose of our notation

```
% PICK VECTOR WITH MOST "ERROR"
pick = findLargeColumn(e);
used = [];
left = 1:q;
W = P(:,pick);
P(:,pick) = []; PP(pick,:) = [];
e(:,pick) = [];
```

The matrix `W` is initialized as the best column of $\mathbf{\Phi}$. The matrices `P` and `e` has their corresponding column removed, `PP` has one element removed. Now update `used` and `left`.

```
used = [used left(pick)];
left(pick) = [];

% CALCULATE ACTUAL ERROR
w1 = p(:,used)';
a1 = radbas(dist(w1,p)*b);
[w2,b2] = solvelin2(a1,t);
a2 = w2*a1 + b2*ones(1,q);
sse = sumsqr(t-a2);
```

19. **Exercise:** Explain each of the previous lines. What is the difference between `W` and `w1`?

We now begin the MAIN LOOP:

```
for k = 1:(q-1)

  % CHECK ERROR
  if (sse < eg), break, end

  % CALCULATE "ERRORS" ASSOCIATED WITH VECTORS
  wj = W(:,k);
  a = wj' * P / (wj'*wj);
  P = P - wj * a;
  PP = sum(P.*P)';
  e = ((P' * d)' .^ 2) ./ (dd * PP');
```

20. **Exercise:** Explain the previous lines of code. To assist you, you may want to refer back to Equation 11.8.

```
  % PICK VECTOR WITH MOST "ERROR"
  pick = findLargeColumn(e);
  W = [W, P(:,pick)];
  P(:,pick) = []; PP(pick,:) = [];
```

```
    e(:,pick) = [];
    used = [used left(pick)];
    left(pick) = [];

    % CALCULATE ACTUAL ERROR
    w1 = p(:,used)';
    a1 = radbas(dist(w1,p)*b);
    [w2,b2] = solvelin2(a1,t);
    a2 = w2*a1 + b2*ones(1,q);
    sse = sumsqr(t-a2);
  end

  [S1,R] = size(w1);
  b1 = ones(S1,1)*b;
```

21. **Exercise:** If you wanted to be able to output the training errors, how would you modify `designrb`?

22. **Exercise:** The following two subfunctions are also included in `newrb.m`. Go through and explain each line.

```
%=======================================================
function i = findLargeColumn(m)

replace = find(isnan(m));
m(replace) = zeros(size(replace));

m = sum(m .^ 2,1);
i = find(m == max(m));
i = i(1);

%=======================================================

function [w,b] = solvelin2(p,t)

if nargout <= 1
  w= t/p;
else
  [pr,pc] = size(p);
  x = t/[p; ones(1,pc)];
  w = x(:,1:pr);
  b = x(:,pr+1);
end
```

23. **Exercise:** Try using RBFs for interpolating a picture. Load the image `clown2.mat`. Think of the image as the graph of a function, then use RBFs to interpolate.

11.7 Homework: Iris Classification

1. Read over and understand the script file attached.

2. Type in the script file.

3. Make appropriate changes so that once the domain set X is loaded, perform a KL reduction to the best three dimensional space. This will be the new domain for the RBF.

4. Run both the original script and the new script, and compare your answers.

11.8 Sample Script File

```
%Script for rbf example on IRIS data

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%The following are handy to name as variables, rather
%than hard-coding them.   NUMITERS is the number of
%steps through the Gaussian width (so we train the
%network that many times).  NUMTRAIN is the number of
%training points we'll use from the data.
%
%Further down, load the data IRIS.NET, and load
%the random indices for the data.  We could call
%randr, but to keep the same random set for more
%than one sessions, the indices were saved.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
numiters=20;
numtrain=120;

X=load('IRIS.NET');
Y=X(:,5:7);
X(:,5:7)=[];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  eg = error goal for the RBF
%  sp = Gaussian spread for the RBF
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

eg=0.1;
sp=linspace(0.05,1.6,numiters);
```

```
load rand001
ii=RandInd;
P=X(ii(1:numtrain),:)';
T=Y(ii(1:numtrain),:)';

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  This suppress those annoying "Your matrix is
%  singular" warnings, which we expect to get here...
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
warning off

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  MAIN LOOP:
%  Train the network on the subsets P and T
%  Simulate the network on the full data set X
%  Get the error in the classification (e)
%  Get the number of weights (c)
%  Get the maximum weight size (W)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


for i=1:numiters
i
net=newrb(P,T,eg,sp(i));
z=sim(net,X');
e(i)=0;

[c(i),b]=size(net.IW{1});

for k=1:150
[x,j]=max(z(:,k));
if Y(k,j)~=1
e(i)=e(i)+1;
end
end
W(i)=max(max(abs(net.LW{2,1})));
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Plotting routines
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

subplot(2,2,1)
```

```
plot3(X(1:50,1),X(1:50,2),X(1:50,3),'r*')
hold on
plot3(X(51:100,1),X(51:100,2),X(51:100,3),'b^')
plot3(X(101:150,1),X(101:150,2),X(101:150,3),'go')
hold off
subplot(2,2,2)
plot(sp,e)
title('Error versus Gaussian Width')
subplot(2,2,3)
plot(sp,c)
title('Number of centers from OLS versus Gaussian Width');
subplot(2,2,4)
semilogy(sp,W)
title('Maximum Weight versus Gaussian Width');
```

Chapter 12

Neural Networks

The term "neural network" has come to be an umbrella term covering a broad range of different function approximation or pattern classification methods.

The classic type of neural network can be defined simply as a connected graph with weighted edges and computational nodes. Let us summarize those we have seen up to this point:

- The Linear Neural Net: $\boldsymbol{x} \to A\boldsymbol{x} + \boldsymbol{b}$.
- Kohonen's SOM: The graph is the network topology, and the units undergo competition.
- Neural Gas: Same interpretation as Kohonen's SOM.
- Radial Basis Functions: Each "center" represents a computational unit, and the output of unit i was $\boldsymbol{x} \to \phi(\|\boldsymbol{x} - \boldsymbol{c}^{(i)}\|)$

In the case of the Linear Neural Net and the RBF, network training was cast as solving a least squares problem, so that we could use features of linear algebra to solve for the best parameters.

In this chapter, we look at the classic workhorse of the neural network industry: The three-layer, feed-forward neural network.

As with the RBF, we will see that the three-layer, feed-forward network is capable of performing function approximation arbitrarily well when the domain is a compact subset of $\mathbb{R}^n$.

12.1 Biological Motivation

The neuron is the basic building block of the central nervous system. There are many different types of neurons, but each can be anatomically broken down into three parts: the dendrites, the cell body, and the axon. Information flows from the dendrites to the cell body through the axon to a synaptic junction connecting to the dendrite to the next neuron.

The synaptic junction is made up of the presynaptic node (the end of an axon), the postsynaptic node (the beginning of a dendrite), and the "empty space", which is the synapse.

For the purposes of our basic mathematical model, we will assume the following processing features that a neuron makes to information (we will refer to Neurons as Nodes in what follows):

1. As information flows across a synapse, information can be amplified, inhibited, or re-polarized. If we let x denote the signal, then synaptic processing is represented as a multiple of x:

$$x \to wx$$

   where w will be referred to as a *weight*. Since there are multiple incoming signals, we denote by w_{ij} the weight connection from Node j to Node i.

2. At the cell body, information from across the dendrites is collected and processed in the following way.

   (a) The incoming signals are summed. Let $j = 1..m$ denote the index of the signal coming to Node i:

$$\sum_{j=1}^{m} w_{ij}x$$

   (b) This quantity is added to the resting state of the cell, which is assumed to be constant:

$$\sum_{j=1}^{m} w_{ij}x + b_i$$

   and b_i is called the bias of the Node. The summed quantity is referred to as the Prestate of the Node.

   (c) Biologically, we assume an "all or nothing" response to the incoming signal. That is, if the signal strength is above a certain threshold, the neuron fires. If not, then the information is not passed along. Mathematically, we use a step function, with the step at 0. This is not a differentiable function, however. We substitute for the step function a "sigmoidal" function, $\sigma(x)$. This is defined to be a monotonically increasing function so that

$$\lim_{x\to-\infty} \sigma(x) = A \qquad \lim_{x\to\infty} \sigma(x) = B$$

   where A and B are constants. Common choices for the sigmoidal function include:

   i. The inverse tangent:

$$\sigma(x) = \operatorname{atan}(x)$$

ii. The exponential sigmoidal function:

$$\sigma_\beta(x) = \frac{1}{1+\mathrm{e}^{-\beta x}}$$

We will use the latter choice, since the computation of its derivative is especially easy to implement on the computer:

$$\sigma'_\beta(x) = \frac{\beta \mathrm{e}^{-\beta x}}{1+\mathrm{e}^{-\beta x}} = \beta\sigma(x)(1-\sigma(x))$$

Other functions that transfer the incoming signal out to the axon are possible. At times, we will use the following, denoted by $t(x)$ as transfer functions:

i. The Linear Node

$$t(x) = x$$

ii. The Circular Node (two inputs, two outputs per node):

$$t(x,y) = \left(\frac{x}{\sqrt{x^2+y^2}}, \frac{y}{\sqrt{x^2+y^2}} \right)$$

iii. The Spherical Node (three inputs, three outputs):

$$t(x,y,z) = \left(\frac{x}{\sqrt{x^2+y^2+z^2}}, \frac{y}{\sqrt{x^2+y^2+z^2}}, \frac{z}{\sqrt{x^2+y^2+z^2}} \right)$$

See [15, 16] for examples of how to implement the last two transfer function types.

3. The axon sends along the information to the synapse.

In summary, we have the following information processing for the i^{th} single node:

$$x \to \underbrace{\sum_{j=1}^{m} w_{ij}x + b_i}_{\text{Prestate}} \to \underbrace{\sigma\left(\sum_{j=1}^{m} w_{ij}x + b_i\right)}_{\text{State}}$$

In vector form,

$$x \to (\boldsymbol{w}_i)^{\mathrm{T}}x + b_i \to \sigma\left((\boldsymbol{w}_i)^{\mathrm{T}}x + b_i\right)$$

So that, taking $\boldsymbol{x} \in \mathbb{R}^N$, and having k Nodes,

$$\boldsymbol{x} \to \left((\boldsymbol{W}^{(0)})^{\mathrm{T}}\boldsymbol{x} + \boldsymbol{b}\right) \to \sigma\left((\boldsymbol{W}^{(0)})^{\mathrm{T}}\boldsymbol{x} + \boldsymbol{b}\right)$$

where

$$\boldsymbol{W}^{(0)} = (\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_k) \quad \boldsymbol{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{pmatrix}$$

and σ operates on a vector or matrix component-wise. Also note that, if σ were a linear function, then the neural network would be linear.

12.2 The Three Layer Feedforward Neural Network

The usual method of describing a neural network's architecture is to use a connected graph with computational nodes. The basic feedforward neural network uses three layers of "nodes". The first layer is the input layer, so the number of nodes is equal to the dimension of $\boldsymbol{x}$, and the transfer function is linear.

The hidden layer contains the nonlinear transfer function, and there is a choice as to how many nodes one includes in the hidden layer. We assume there are k nodes in the hidden layer.

The output layer again has a linear transfer function, and there are as many nodes here as there are output variables. Each edge of the graph is weighted, and we refer to edges from the input layer to the hidden layer as the matrix

$$\boldsymbol{W}^{(0)} = \left[W_{ij}^{(0)}\right]_{i=1:n(1),\; j=1:n(0)}$$

where $n(i)$ is the number of nodes in Layer i, and

$$\boldsymbol{W}^{(1)} = \left[W_{ij}^{(0)}\right]_{i=1:n(2),\; j=1:n(1)}$$

so that W_{ij}^{k} refers to the weight in the k^{th} layer, going from Node j in the k^{th} layer to Node i in the $k+1^{\text{st}}$ layer.

There are also vectors of biases associated with the hidden layer, and output layer, respectively:

$$\boldsymbol{b}^{(0)} \text{ and } \boldsymbol{b}^{(1)}$$

For the purposes of the backpropagation algorithm given later, note that the biases can also be represented as one additional node in the input and hidden layers, with constant transfer function, $\sigma(x) = 1$. We will adopt this notation, and suppress the bias specification in Section 4.2.1. Figure 12.1 shows the graphic layout of the three layer, feedforward neural network.

The purpose of this network is to provide an approximation to an arbitrary continuous function, $\boldsymbol{F}$, defined on a compact domain X, and we will use the notation $\mathbf{y} = \boldsymbol{F}(\mathbf{x})$, where $\mathbf{x} \in \mathbb{R}^N$ and $\mathbf{y} \in \mathbb{R}^M$. The network will render the

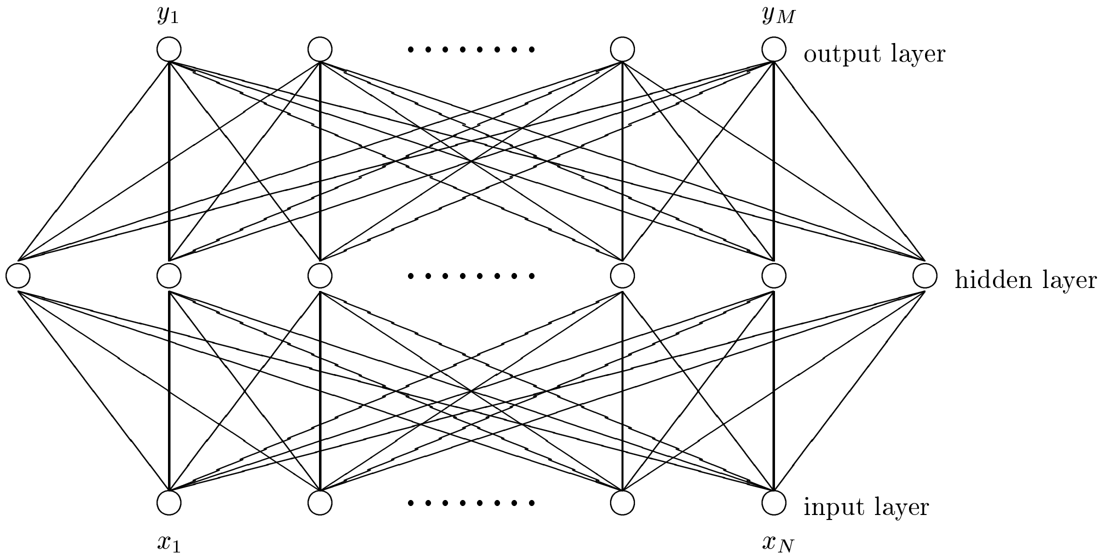

Figure 12.1: The neural network architecture.

approximation in terms of a sigmoidal basis so that the approximation to $\boldsymbol{F}$ is given pointwise by:

$$\boldsymbol{y}^{(i)} = \boldsymbol{F}(\boldsymbol{x}^{(i)}) \approx \left(\mathbf{W}^{(1)}\sigma\left(\mathbf{W}^{(0)} \cdot \mathbf{x}^{(\mathbf{i})} + \mathbf{b}^{(0)}\right) + \mathbf{b}^{(1)}\right) \tag{12.1}$$

In a paper of fundamental importance, Cybenko [7] shows that sums of the form in equation (12.1) are dense in the space of continuous functions defined on the unit cube. In other words, a neural network is capable of approximating any continuous function arbitrarily well by increasing the number of nodes in the hidden layer. As the number of nodes in the hidden layer increases, the error between the network approximation and the continuous function goes to zero. This is very significant for several reasons:

- It gives us a theoretical framework from which to work.
- The proof of this theorem ties neural networks to the classical approaches to function approximation in mathematics.
- It solves one of Hilbert's problems from the turn of the last century. Kolmogorov had given the first proof in the 1950's, but it is still not clear that the functions he used can be implemented in a computer algorithm.
- The proof revitalized neural network research. Prior to this, research had been stagnant after the discovery that the first neural networks (in the 1960's) were incapable of modeling certain standard functions (the XOR function). We know now that this was because the first networks only worked on linearly seperable data.

- Although we have taken outrageous liberties (biologically speaking) in our model, the model shows that a very simple structure is capable of very complex behavior.

In fact, there are more general results that can be found in [13, 14, 11, 12, 9, 10] and more specifically, they show that a neural approximation not only approximates the function, but can also simulatenously approximate the first m derivatives using the same functional parameters.

12.3 Training and Error

Theoretically, we have now seen the neural network as a "Universal Function Approximator". The real question now is, given an input and output set, how does one build the neural net? This question can be reframed as: Find the weights w^i_{jk} and biases b^i_j that perform a desired input-output. This process is also known as network training.

For a fixed value of the parameters in the matrices $\boldsymbol{W}^{(i)}$ and biases $\boldsymbol{b}^{(i)}$, the network outputs a smooth function in $\boldsymbol{x}$. As mentioned previously, we adopt the notation of including the biases in the parameters $\boldsymbol{W}^{(i)}$, and add an extra node on the hidden and input layers.

Define the mean square error of the approximation, given a set of parameters $\{\boldsymbol{W}^{(0)}, \boldsymbol{W}^{(1)}\}$, as:

$$E(\boldsymbol{W}^{(0)}, \boldsymbol{W}^{(1)}) = \frac{1}{2|X|} \sum_{i=1}^{|X|} \|F(\mathbf{x^{(i)}}) - \left(\mathbf{W}^{(1)}\ \sigma\left(\mathbf{W}^{(0)} \cdot \mathbf{x^{(i)}}\right)\right)\|_2^2 \qquad (12.2)$$

where $|X|$ denotes the number of points in the data set X. Then E is a smooth function of the weights and biases, so that it can be minimized. From the method of steepest descent, we update the weights:

$$W^{(i)}_{jk}(t+1) = W^{(i)}_{jk}(t) + \alpha(t)d(t) \qquad (12.3)$$

until the error has become smaller than a prescribed value. For the standard gradient descent, for example, we have

$$d(t) = \frac{\partial E}{\partial W^{(i)}_{jk}}(t)$$

How we find the value of $\alpha(t)$ and the definition of $d(t)$ is where the divergence in training algorithms occurs. There are many algorithms built especially for this purpose, such as Line Minimization. Other update methods can be applied, since training is a general, unconstrained optimization problem:

$$\min_{\boldsymbol{W}^{(0),(1)}} E$$

Most optimization algorithms will require that we obtain expressions to compute the partial derivatives of E with respect to the weights and biases. It is fortunate that, for large networks, there is a recursive algorithm to accomplish this. This will be discussed in the next section. First, we introduce some notation that will make the algorithm more clear.

Define the prestate for layer i, Node j at time t:

$$P_j^i = \sum_{k=1}^{n(i-1)} W_{jk}^{i-1} S_k^{i-1}, \text{ for } \; i = 1 \ldots L$$

where S_j^i is the State of Node j on layer i, which is

$$S_j^i = \begin{cases} \sigma\left(P_j^i\right) & i > 0 \\ x & i = 0 \end{cases}$$

12.3.1 Backpropagation of Error

We now detail a recursive algorithm, known as the backpropagation of error [1], which gives an efficient method for computing the partial derivatives for training (see Equation (12.3)).

The goal is to compute

$$\frac{\partial E}{\partial W_{jk}^{i-1}}$$

for each i, j, k.

From the chain rule, we can write

$$\frac{\partial E}{\partial W_{jk}^{i-1}} = \frac{\partial E}{\partial P_j^i} \cdot \frac{\partial P_j^i}{\partial W_{jk}^{i-1}}$$

which we can re-express, using:

$$\frac{\partial P_j^i}{\partial W_{jk}^{i-1}} = S_k^{i-1} \qquad \frac{\partial P_j^{i+1}}{\partial P_k^i} = W_{jk}^i \sigma'(P_k^i)$$

and defining:

$$\delta_j^i = \frac{\partial E}{\partial P_j^i}$$

Then we have:

$$\frac{\partial E}{\partial W_{jk}^{i-1}} = \delta_j^i \cdot S_k^{i-1}$$

So, we need an algorithm to compute δ_j^i.

For the output layer (L), with aribtrary transfer function σ,

$$\delta_j^L = \frac{\partial E}{\partial P_j^L} = \frac{\partial E}{\partial S_j^L} \frac{\partial S_j^L}{\partial P_j^L} = \sigma'(P_j^L) \frac{\partial E}{\partial S_j^L}$$

where $\frac{\partial E}{\partial S_j^L}$ is computed from the error by substituting S_j^L into Equation (12.2):

$$E = \frac{1}{|X|} \sum_{i=1}^{2|X|} \|\boldsymbol{y}^{(i)} - \begin{pmatrix} S_1^{(L)} \\ \vdots \\ S_m^{(L)} \end{pmatrix} \|_2^2$$

which implies that:

$$\frac{\partial E}{\partial S_j^L} = \frac{-1}{|X|} \sum_{i=1}^{|X|} |y_j^{(i)} - S_j^L| \tag{12.4}$$

For the hidden nodes,

$$\delta_j^i = \frac{\partial E}{\partial P_j^i} = \sum_{k=1}^{n(i+1)} \frac{\partial E}{\partial P_k^{i+1}} \frac{\partial P_k^{i+1}}{\partial P_j^i}$$

Noting $\frac{\partial P_k^{i+1}}{\partial P_j^i} = W_{kj}^i \sigma'(P_j^i)$, we obtain:

$$\delta_j^i = \sigma'(P_j^i) \sum_{k=1}^{n(i+1)} \frac{\partial E}{\partial P_k^{i+1}} W_{kj}^i$$

Which, from our definition of δ_j^i, is recursively defined as:

$$\delta_j^i = \sigma'(P_j^i) \sum_{k=0}^{n(i+1)} \delta_k^{i+1} W_{kj}^i \tag{12.5}$$

This can be interpreted as a backpropagating the δ_j^i backwards through the network. The algorithm is summarized as:

Algorithm 12.3.1 *The Backpropagation Algorithm*

1. Forward propagate all the x's, keeping track of the error.
2. Compute δ_j^L, for $j = 0 \ldots M-1$ using:
$$\delta_j^L = \sigma'(P_j^L) \frac{\partial E}{\partial S_j^L}$$
3. Compute all other δ's:
$$\delta_j^i = \sigma'(P_j^i) \sum_{k=0}^{n(i+1)} \delta_k^{i+1} W_{kj}^i$$
4. Compute:
$$\frac{\partial E}{\partial W_{jk}^{i-1}} = \delta_j^i \cdot S_k^{i-1}$$
for $i = 1 \ldots L$, $j = 1 \ldots n(i+1)$, and $k = 1 \ldots n(i)$.

12.3.2 Nonlinear Optimization Techniques

There are many methods to choose from, and some are included below:

- Steepest Descent (with variations on how to compute α)
- Newton's Method (This is indirect - we solve for where $f'(x) = 0$).
- Conjugate Gradient (Search along the eigenvectors of the Hessian).
- Levenburg-Marquardt (A combination of the methods above).

To really study all of these techniques would take us too far afield in this course. Any text on nonlinear optimization (as well as "Numerical Recipes for C") will cover the basic ideas.

The introduction of a nonlinear optimization technique is the primary reason we will use Matlab to perform our neural network training. Programming these techniques can be a delicate task! For our purposes, Levenburg-Marquardt is more than adequate.

12.4 Neural Networks and Matlab

Matlab has set up a very general data structure by which to construct and train almost any type of neural network. Here we focus on the construction and operation of the three-layer feed-forward network.

Because the data structure is so general, building and training a neural network is broken into multiple stages.

- Identify the network architecture. This includes constructing neural layers, connections, identifying the transfer functions for each node. Furthermore, one must define the error measure (or performance index), the training function, and how one wishes to compute a Jacobian matrix. This step would be extremely tedious if we had to do it every time! Matlab has several built in architectures, and we've dealt with a few. For example:

  - `newsom` initializes Kohonen's SOM.
  - `newlin` initializes a linear neural network.
  - `newrb` initializes a radial basis function network.
  - `newrbe` initializes (and trains) an exact RBF.

  The new function here is `newff`.

- Train the network. This is performed (for all network types) by the command: `net=train(net,X)`. The specific functions and training parameters have already been set up by the initialization process.

- Simulate the network with new data. Again, this is performed generically by: `Y=sim(net,X)`. The specific functions and parameters have been set in the network data structure.

Before continuing, lets look at an example:

Sample Training Session 1:

```
1 P = [0 1 2 3 4 5 6 7 8 9 10];
2 T = [0 1 2 3 4 3 2 1 2 3 4];
3 net = newff([0 10],[5 1],{'tansig' 'purelin'});
4 Y = sim(net,P);
5 plot(P,T,P,Y,'o')
6 net.trainParam.epochs = 50;
7 net = train(net,P,T);
8 X = linspace(0,10);
8 Y = sim(net,X);
9 plot(P,T,'ko',X,Y)
```

Training Session 1 explanation:

- Lines 1 and 2 set up the training patterns and targets. Notice that the domain runs between a minimum of 0 and a maximum of 10, which are the numbers appearing in the first argument of `newff`. Notice also that the sizes of the domain and range sets are given as $m \times n$, where m =dimension and n =number of data points.

- Line 3 initializes a feed forward neural network. The arguments shown are required, and later we will see what other optional arguments are allowed. For now, notice that the first argument gives the minimum and maximum values in the domain set. Normally, one uses `minmax(P)` in place of actually typing these values in. This also implicitly defines the number of nodes in the input layer (which is the domain dimension).

  The second argument in Line 3 defines the number of nodes in the hidden layer and in the output layer. This vector also implicity defines how many layers you have by counting the size of this vector. Thus, we can have as many layers as we wish.[1] Later, we will define a network with multiple hidden layers.

  The last required argument in Line 3 is the type of transfer function that will be used. The `tansig` function is the inverse tangent function, which gives the same performance as our standard sigmoidal function. We will always use a linear layer in the output, so this argument is always `purelin`.

- Lines 4 and 5 show you what the neural network output looks like before network training (the weights were initialized using `initwb`).

- Line 6 sets up how many training iterations (or epochs) will be performed. This means that the network will run through all of the data 50 times.

[1] Theoretically, we require only one hidden layer. However, many researchers find it easier to train a network with two small hidden layers versus one large hidden layer. This is mainly a matter of taste.

- Line 7 trains the network using Levenburg-Marquardt (which is the default).
- Lines 8 and 9 simulate the network and plots the result. Note that we plot more than just the trained points to see if the network generalizes well.

Exercise:

1. Run the sample exercise as written and plot the output.
2. Increase the number of epochs to 1000 and retrain and replot.
3. Create a new network with 10 nodes in the hidden layer, train using 600 epochs. Plot the result.
4. Compare the plots and discuss the results.

The NEWFF command

The full `newff` command looks like:

```
net = newff(M,[S1 S2...SN],{T1 T2...TN},F1,F2,F3)
```

Description of the input parameters:

- M is a vector of minimum and maximum values of the data. The size of M will be $n \times 2$, where n is the dimension of the input data (and so will define the number of nodes in the input layer). We will almost always use the function `minmax(P)` for this argument.
- The next vector lists the number of nodes in the next N layers (not the input layer). Thus, for a three layer network, we will always use $[a \quad b]$ where a is the number of nodes in the hidden layer, and b is the output dimension.
- The next list defines the transfer function for each layer. We will almost always use `{'tansig','purelin'}`, although other options are available. In fact, below we will write our own transfer function.
- The next three functions define (in order):
  - `F1` is the nonlinear optimization function to use. The default is Levenburg-Marquardt (`trainlm`). Others are possible. Type `help newff` to read the WARNING message about memory requirements and different possible options.
  - `F2` defines how the Jacobian matrix will be computed. The default function is `learngdm` which computes the Jacobian via backpropagation.
  - `F3` is the performance function, whose default uses the mean squared error (`'mse'`), which we will use.

12.4.1 Training Notes

In this section, we look at how to train a neural network, and what to do if the training is not progressing well. What constitutes a good network? How do we choose the number of nodes in the hidden layer?

Parameters and Preprocessing

What may be striking in the `newff` command is the absence of any training parameters. Recall that in the section on Radial Basis functions, there was a parameter that controlled the width of the Gaussians. However, also recall that changing the widths of the Gaussians had the same effect as rescaling the data. That is what we may have to do when using a feed forward network. Let's see an example.

In Figure 12.2, we see a standard sigmoidal function. For small values of x, the corresponding images, $\sigma(x)$ are quite different. We say that σ is able to differentiate between these data points. On the other hand, suppose that most of your data is out at the "tails" of the sigmoidal. Then the corresponding images begin to pile up at the horizontal asymptotes. That is, for example, that $\sigma(19)$ becomes indistinguishable from $\sigma(50)$. For example, if $\sigma(x) = \frac{1}{1+e^{-x}}$, then a calculator might say that $\sigma(19) = 0.9999999$, and $\sigma(50) = 1$. Therefore, if all of the data is between 19 and 50, the output will almost always be 1.

This effect is called **saturation**- we say that the sigmoidal function has been saturated if the data has not been scaled appropriately.

This is an effect to keep in mind if the neural network is not training very well. Rather than having a lot of training parameters in the function call, we will assume that the data has been scaled "appropriately". In other words, we will always look at preprocessing the data before giving it to the neural network to train. Let's look at some options in preprocessing that Matlab gives us. Each method has three associated commands. One command is to calculate the processing parameters from the given data and to perform that method on the data (the method begins with `pre`), another command is for postprocessing the data (the inverse of the preprocessing function, and begins with `post`), and a third command that will process new data using parameters that have already been computed (for sending new data to the network, and begins with `tra`).

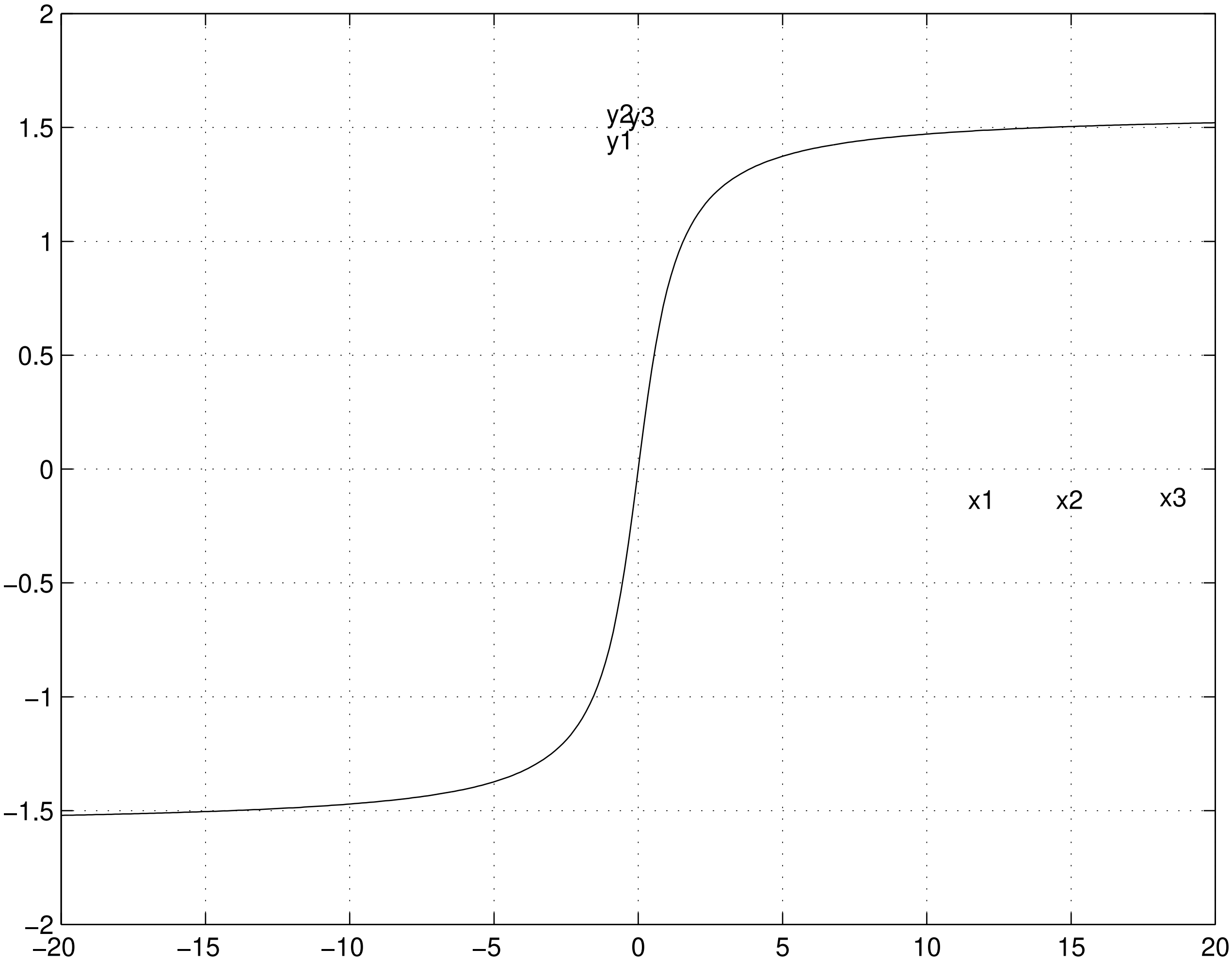

Figure 12.2: The Sigmoidal Function, and an example of "Saturation": As the size of the input values gets large, σ loses its ability to distinguish between data points.

prestd	Normalize data for zero mean, stand. dev. 1
poststd	Undo the PRESTD command
trastd	Process new data using the precalculated parameters.
premnmx	Normalize data for maximum of 1 and minimum of -1.
postmnmx	Undo the PREMNMX command.
tramnmx	Transform new data with precalculated parameters.
prepca	Principal component analysis on input data with zero mean.
	Matlab doesn't provide a post processor.
trapca	Transform data with PCA matrix computed by PREPCA.

Command Input/Output Arguments:

```
[pn,meanp,stdp,tn,meant,stdt] = prestd(p,t)
[p,t] = poststd(pn,meanp,stdp,tn,meant,stdt)
[pn] = trastd(p,meanp,stdp)

[pn,minp,maxp,tn,mint,maxt] = premnmx(p,t)
[p,t] = postmnmx(pn,minp,maxp,tn,mint,maxt)
[pn] = trastd(p,minp,maxp)

[ptrans,transMat] = prepca(P,min_frac)
[Ptrans] = trapca(P,TransMat)
```

Exercise: Write commands prekl, postkl and trakl that are similar to the pca commands, but also performs mean subtraction. You'll need to decide on input/output arguments.

Exercise: Write commands premax, postmax, and tramax that transform a data set so that it has zero mean and a maximum absolute value (in each dimension) of 1. Again, you'll need to decide on input/output arguments.

Network Complexity and Generalization

By the theorems we stated earlier, it is possible for a three layer neural net to approximate an arbitrary continuous function over a compact domain to an arbitrary precision. That is, if we added enough nodes in the hidden layer, we could get arbitrarily close to the function. By the way, in the proof of such statements, it is assumed that there is enough data to accurately reproduce the function to arbitrary accuracy.

So adding new hidden nodes in a three layer network is akin to adding new centers to an RBF. Thus, we begin to run into the same old problem in the approximation of a function: The tradeoff between accuracy on the training set, and generalization off the training set.

The rule of thumb in getting a good neural net is to use the least number of hidden nodes that you can to still get the accuracy that is desired.

Nice Demonstration: In Matlab, run the demonstration nnd11gn. This will animate the network response as the training algorithm proceeds.

An alternative procedure that I would not recommend, but other researchers do is to use plenty of hidden nodes, but to stop training early. The reason I don't like this option is that the more hidden nodes one uses, the higher the dimension of the error space. For example, if the input dimension is n and the output dimension is m, then adding another hidden node adds $n + m + 1$ extra training parameters, and so increases the dimension of the error space by $n + m + 1$. It is common in network training to get stuck in local minima. By increasing the dimension of the error space, it is my opinion that you increase the chances of getting stuck (not to mention that you increase the amount of computation and memory required to train).

One way that we might think about the number of nodes in the hidden layer is to consider that the first part of the mapping (before the sigmoidals are applied) is a linear transformation from $\mathbb{R}^n$ to $\mathbb{R}^k$, where the input dimension is n and we have k hidden nodes. Heuristically, we need enough nodes in the hidden layer to unfold the data in such a way as to allow the sigmoidals to differentiate between data points (both in terms of saturation and in keeping the projection $1 - 1$).

12.5 Post Training Analysis

How do we determine whether or not our network is doing a good job? Below we consider some options for testing the neural network.

1. Test how the network works on the Validation Set. The validation set is there as a check to make sure we're not overtraining the neural network. One way to use the validation set is to stop training and check the error on the validation set. The idea is that initially the error on both the training and validation sets will decrease, but as the network begins to overtrain (if it does), then the error on the validation set will increase. If this occurs, we stop training.

   Matlab has this utility built into its `train` function.

   An example:

```
%Create training set
p=[-1:0.05:1];
t=sin(2*pi*p)+0.1*randn(size(p));

%Create a validation set
v.P=[-0.975:0.05:0.975];
v.T=[sin(2*pi*v.P)+0.1*randn(size(v.P))];
```

```
%Create a network, and train using the validation
net=newff(minmax(p),[20,1]);
net.trainParam.show=25;
net.trainParam.epochs=300;
[net,tr]=train(net,p,t,[],[],v);
```

Here we also see the added output argument to `train`, a vector `tr`, which is a training record of the errors. The added empty vectors are for "input delay" arguments, which we will not use in this course.

When run, the training should stop early due to an increase of the error on the validation set.

2. A second method is to perform some analysis on the errors that the network produces. There are several methods to do this (including some statistics that are beyond the scope of this course). We discuss three commonly used procedures:

   (a) Matlab has a `postreg` function to analyze errors for 1-dimensional output. This function performs a linear regression between the neural output of the function and the target values. The general idea is that the higher the correlation, the better the training for the network. Here is an example:

   ```
   p=[-1:0.05:1];
   t=sin(2*pi*p)+0.1*randn(size(p));

   v.P=[-0.975:0.05:0.975];
   v.T=[sin(2*pi*v.P)+0.1*randn(size(v.P))];

   net=newff(minmax(p),[10,1]);
   net.trainParam.show=25;
   net.trainParam.epochs=200;
   [net,tr]=train(net,p,t);
   an=sim(net,v.P);
   [m,b,r]=postreg(an,v.T);
   ```

   with the resulting graph and correlation as shown in Figure 12.3.

   (b) A second method is taken from the idea that if the training is good, then the error should be uncorrelated. Put another way, all correlations should be explained by the neural network model. We can therefore look at the correlations in the error. Here's an example:

   ```
   x=3*rand(500,3)-1;
   y(:,1)=x(:,1).^2-x(:,2);
   y(:,2)=y(:,1)-exp(x(:,3));
   y(:,3)=sin(x(:,1));
   net=newff(minmax(x'),[20,3],{'tansig','purelin'});
   ```

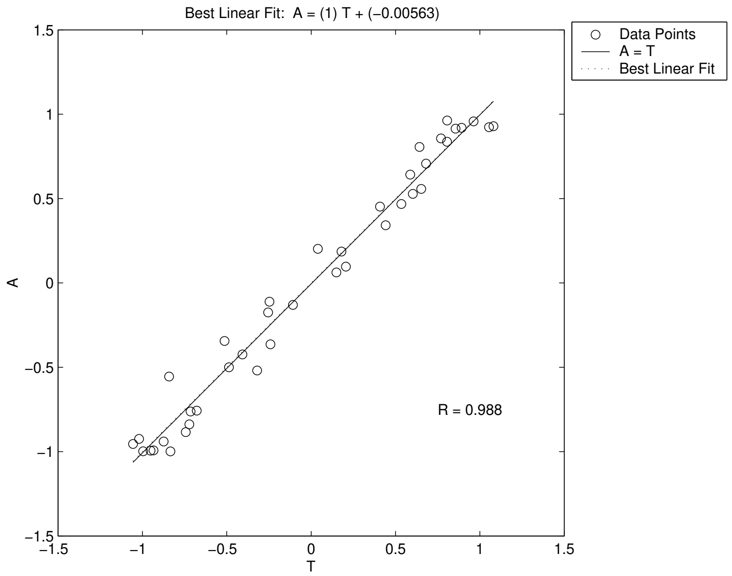

Figure 12.3: The result of using post regression on the targets. Obtaining a correlation as shown means that the network has trained well.

```
net.trainParam.epochs=75;
net=train(net,x',y');
vp=3*rand(200,3)-1;
vt(:,1)=vp(:,1).^2-vp(:,2);
vt(:,2)=vt(:,1)-exp(vp(:,3));
vt(:,3)=sin(vp(:,1));
an=sim(net,vp');
G=an-vt';
[U,S,V]=svd((1/200)*G*G');
S=
   1.0e-06 *

    0.3983         0         0
         0    0.3534         0
         0         0    0.0042
```

(c) The last method we discuss is similar to the second. We can examine the histogram of the size of the errors. If training went well, we can expect the errors to be clustering about zero, and rapidly trail off.

3. Examine the weights of the network. Overtraining will generally lead to very large weights (recall what some of the weights looked like in the RBF network). Extremely large weights are undesireable, because they can lead to instabilities in the network- i.e., small deviations in the domain can lead

to huge deviations in the range.

Similarly, weights that are extremely small may suggest that training could have been done with fewer nodes in the hidden layer.

4. Regularization. A detailed explanation of regularization is beyond the scope of these notes, but we can give some heuristics. The basic idea is to penalize those weights that get very large in favor of small weights. Formally, we change the error function. Let E denote our standard mean squared error. Assume all training parameters (weights and biases) are indexed as: W_i, $i = 1, 2, \ldots, N$. The new error function, $\mathrm{MSE}_{\mathrm{reg}}$, is defined as:

$$\mathrm{MSE}_{\mathrm{reg}} = \gamma E + (1-\gamma)\frac{1}{N}\sum_{i=1}^{N} W_i$$

where γ is called the *regularization parameter*. As we can see, if $\gamma = 1$, the weights play no role in the training, but if $\gamma = 0$, the weights play the only role in the training. In general, the smaller the parameter, the more smooth the approximation.

The problem is in knowing what to make the regularization parameter. If it is too large, the network may overfit. If it is too small, we may not be modeling the data very well. Before considering automatic methods to determine the regularization parameter, the following example shows how to implement this new error:

```
net=newff(minmax(X),[5,1]);
net.performFcn='msereg';
net.performParam.ratio=0.5 %This is gamma
```

Matlab has a Bayesian regularization technique installed that tries to use Bayesian statistics to determine the best value of γ. In addition, it outputs a value called "effective number of parameters" of the network. If the effective number of parameters is far less than the number of parameters used, it suggests that you use a simpler network for training. Here is an example of its use:

```
net=newff(minmax(X),[5,1],{'tansig','purelin'},'trainbr');
```

That is, the `trainbr` stands for training with Bayesian regularization.

12.6 Example: Alphabet recognition

In this project, we look at how a neural network can be used for character recognition. This is extremely useful for companies such as the U.S. Postal Service, and they actually use neural networks to accomplish the reading of address information.

We will do a simpler example. We will assume that a letter is represented by a 5×7 matrix of values that are either 1 or 0. Therefore, the domain of the neural network will be 35 dimensional.

The output of the neural network will be 26-dimensional, where there is a 1 in the numerical position corresponding to its letter.

Our goal: Character recognition, even when the characters have been contaminated by noise[2].

Architectural Items

- We will utilize a $35 - 10 - 26$ feed forward neural network using a `tansig` transfer function for both the hidden and output layers.
- We will use a standard gradient descent to speed up training, `traingdx` in place of the default `trainlm`.
- We will use a sum of squares error `sse` in place of the usual mean square error `mse`.
- The training error goal should be set to 0.1.
- Use a maximum of 5000 training epochs.
- Additionally, we need to set a momentum term for the gradient descent by typing: `net.trainParam.mc=0.95`

The Example

Run the example `appcr1`. Print and analyze the file `appcr1.m`. Annotate each line of code to understand what is occuring.

12.7 Project 1: Mushroom Classification

In this project, we use a neural net to perform classification on mushroom characteristics. That is, the neural net should input several characteristics of a mushroom, and output whether or not that mushroom is poisonous or edible.

The data consists of a matrix of 4062 entries each containing 22 characteristics of a mushroom (i.e., cap size, cap shape, stalk color, etc.). The range is either 0 or 1, depending on whether the mushroom is edible or not. Obtain this data by loading `mushroom_pre.mat`.

Project Questions:

1. Discuss any problems there might be in using an RBF network.

[2] In real world systems, the contamination is not by noise but by people's sloppy handwriting!

2. Decide on how you are going to preprocess the data. Run a global KL on the data to see how it is sitting in high dimensional space. Plot the best two dimensional representation.
3. Retain 1,000 points of data for training, and reserve the rest for validation of the network model. Use as few nodes in the hidden layer as necessary in order to get an error of 0 in classification.
4. Plot the results of the training. That is, first plot the network results for those mushrooms corresponding to class 1 as blue dots. Plot the network output for those mushrooms corresponding to class 0 as red dots. You should see a wide seperation between red and blue.
5. Does preprocessing the data help? Rerun the network training, except now use enough dimensions to retain 90% of the energy. Use the projected data for the domain of your network.
6. Turn in the script file for training the two networks, and any plots you used.

12.8 Autoassociation Neural Networks

In the previous section, we saw that a three layer feedforward neural network is capable of approximating any continuous function on a compact domain.

Suppose now that we want to reduce the dimensionality of the data set X. This can be expressed as constructing homeomorphisms

$$G : X \subset \mathbb{R}^N \to Y \subset \mathbb{R}^M$$
$$H : Y \subset \mathbb{R}^M \to X \subset \mathbb{R}^N$$

We could build two separate neural networks, one for each function, if we knew a priori the target values in the reduced dimensional space Y. For the moment, suppose we are not concerned with the form of the reduced representation. Then the compression paradigm can be reformulated: find functions G and H as before, but we only require that the composition $G \circ H$ is the identity on X. We envision this composition as a single, large network composed of two, three layer networks glued together, as shown in Figure 12.4.

In this figure, the input and output layers have N nodes, the bottleneck layer has $M << N$ nodes, and the two hidden layers have some pre-determined number of nodes. It has been shown in [7, 10, 28] that these hidden layers are necessary to perform a nonlinear compression and decompression. That is, without the nonlinear hidden layer, this network performs a linear dimensionality reduction- for which we know the optimal solution is given by its Karhunen-Loève expansion.

This network has been widely used as a compression technique, and in a recent work [17] several criteria were included in the training algorithm to force certain specific representations on the bottleneck layer.

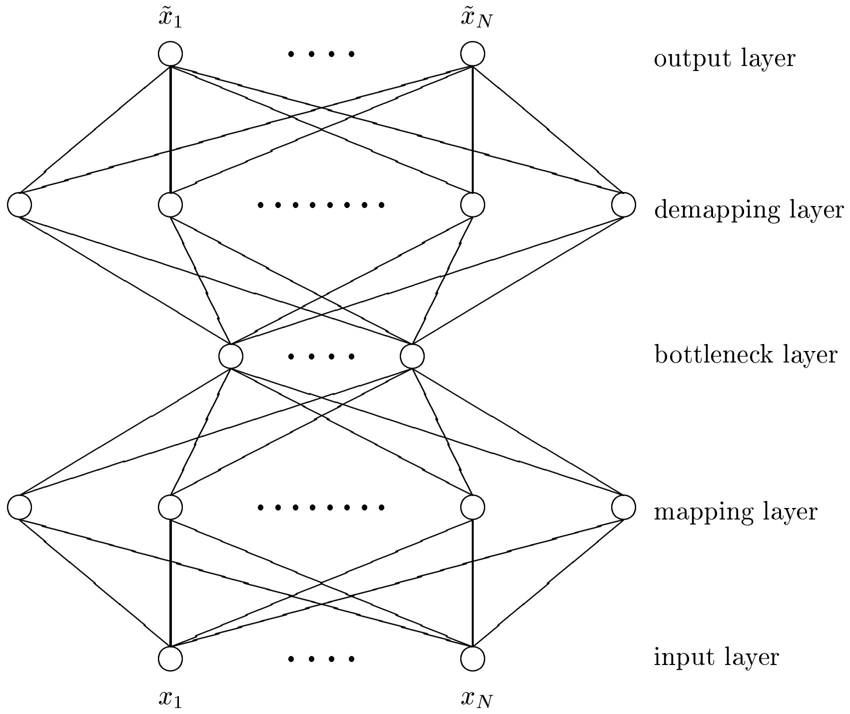

Figure 12.4: The bottleneck network architecture.

Matlab and the Autoassociator

In Matlab, it is simple to implement an autoassociative network, since we can define any number of hidden layers. In this case, we have a 5 layer network, so that if the input dimension is n, the hidden layer dimension is k, and the target dimension is m, then the command:

```
net=newff(minmax(P),[k,m,k,n],{'tansig','tansig','tansig','purelin'});
```

will construct the network. Notice that the domain data is the same as the range data.

Question: Why will the neural network not simply construct an identity function, $f(x) = x$?

Project: Unknotting a knot